Embedded Programs

EMBEDDED TECHNIQUES

John Dybowski

All embedded controllers must have their main program in some sort of nonvolatile memory, whether it be EPROM, EEPROM, Flash memory, or battery-back RAM. Which you choose depends on the application.

Embedded computers generally run under the control of stored programs. Most often these programs are stored in EPROMs where they are, for all practical purposes, permanent. Although burning programs into these "stone tablets" is pretty much the standard mode of operation, there are several issues associated with this practice.

For anyone charged with maintaining a battery of embedded instruments, the primary concerns revolve around the amount of time and effort involved in changing EPROMs if, or when, the need arises. Even if you've specified the application accurately and your code is perfect in every way, it's not unreasonable to consider that the application itself may change over time. I'll be the first to admit that there are a great many applications that can be served with a fixed and unchanging feature set. Then again, if you're designing anything that people interact with, odds are system requirements will grow and evolve as users' needs and expectations change. Anyway, whether your program update is planned or unplanned, and whether it's to add features or to fix bugs, it quickly becomes apparent that wrestling with EPROMs is less than optimal for promoting happy customers.

PLIABLEWARE

If you don't want to rip your system apart every time a code change is required, then you need to store your program in something other than a conventional EPROM. You would be better off with something that is disposed to in-circuit programming. Remember that, by their very nature, embedded computers usually get crammed into out-of-the-way places and often exist in large numbers. These are two good reasons why changing chips tends to be a costly and time-consuming affair. Such hidden expenses are becoming an increasingly important concern as evidenced by more and more customers that are willing to pay extra up front to avoid such entanglements. To satisfy these changing product needs, chip manufacturers are supplying a wide variety of alternate memory options.

Flash technology is perhaps one of the most familiar and is rapidly gaining favor with many engineers. Although differing in detail, the closely related E²PROM devices are also useful in many of the same applications. With the right support circuitry, even battery-backed RAM can be used to satisfy these same purposes. But, regardless of the technology used, it is advisable to make sure an appropriate write protection technique can be applied. It goes without saying that for an embedded computer to erase its executable program would be catastrophic...and inexcusable.

Early E²PROMs addressed this problem somewhat indirectly by requiring various weird voltages to enable the programming function. This "security" feature carried over to Flash devices as many of these parts still require a "high voltage" programming supply. The arrival of Flash and E²PROM devices capable of programming using just the standard logic power supply resulted in a reliance on software-based write-protection algorithms.

Updating program memory in-circuit is, in many applications, the way to go. This can be done using a purely hardware approach where you directly seize control of the memory device by tapping onto its pins using some sort of special programming connector. An alternative, and more attractive, method would allow doing the programming operation under firmware control. Obviously with this approach you'd have to keep at least a

little code on-line so you can acquire and load the new program from the system controller. One way of doing this is to use a small dedicated EPROM that holds your program loader and communications handler.

Another way you could set up your system would be to use an E²PROM where your download and communications utilities could be kept in a separate, out of the way, part of the chip. Apart from the technological differences, one of the more apparent dissimilarities between an E²PROM and a pure Flash device is that the E²PROM can be erased and programmed on a byte-by-byte basis whereas the Flash part must be completely erased before it can be reprogrammed. In this respect it functions just like a standard EPROM except that the erasure is invoked electrically instead of by using ultraviolet light. Needless to say, having to clear the entire Flash device is bothersome since it mandates a separate storage device for the code that must be permanently maintained.

In response to this objection, sectored parts are now available which contain multiple virtual Flash devices in various groupings on one chip. Some of the so-called "boot block" devices have what essentially amounts to an EPROM sector that can only be programmed by applying a high programming voltage to the chip. That is, these "boot sectors" can only be programmed and erased using device programmers and cannot be altered once they are installed in the system. This provides the needed security for critical low-level code and eliminates the need for a separate EPROM.

These developments illustrate the various types of nonvolatile storage and their associated stages of security. Even something as conceptually straightforward as nonvolatile memory actually exists in degrees. The distinctions between technologies tend to blur, however, and with the addition of a partitioned nonvolatile controller like the Dallas DS1610, even a static RAM chip can take on some of the attributes of E²PROM and Flash parts. Generally speaking, all of these parts are capable of doing the same thing. And as so often happens in electrical engineering, the details actually determine which technology is appropriate (or most appropriate) for a given task.

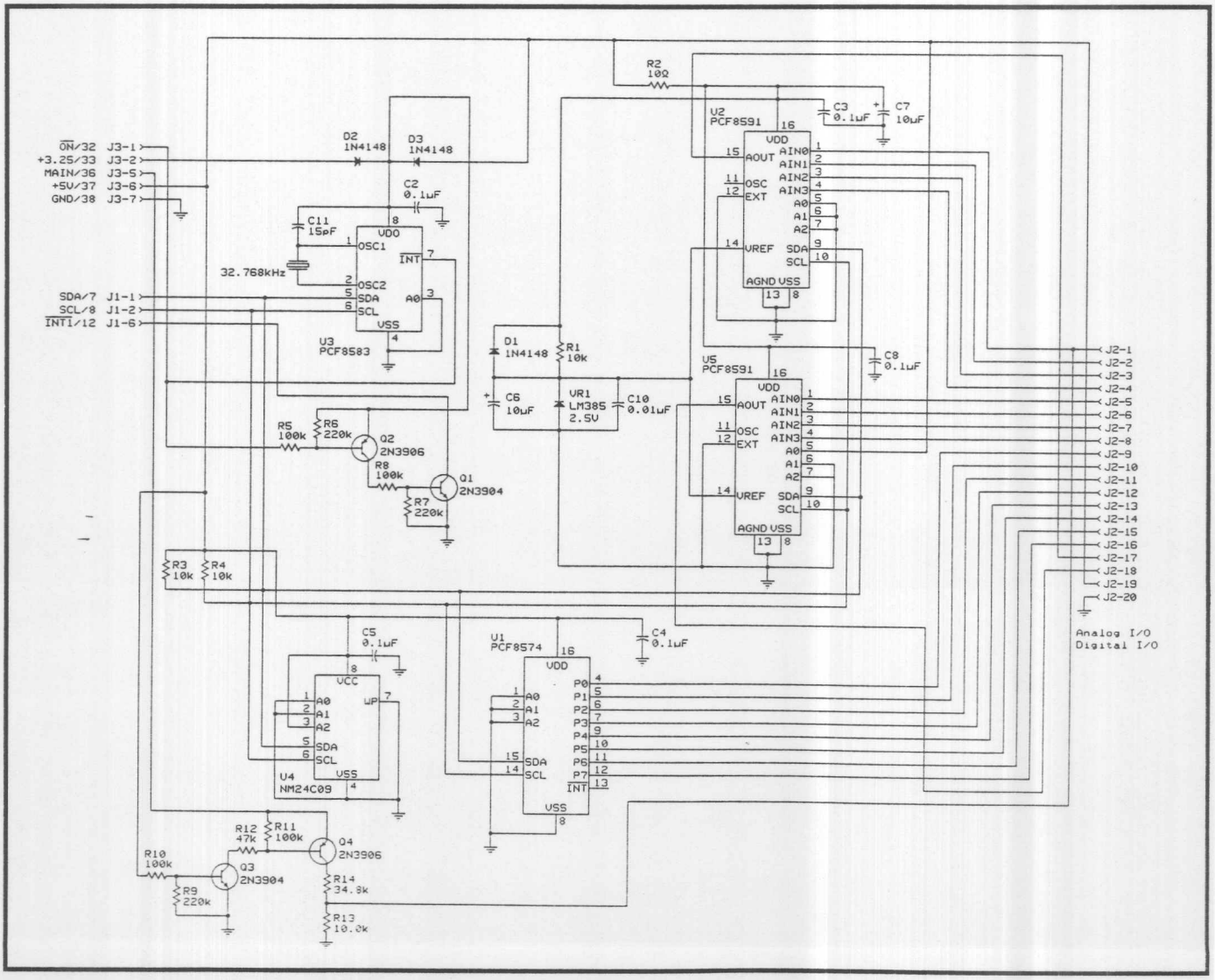

Figure 1—*All circuitry including the PCF8583 RTC is located on the I²C card, however, back-up power for the RTC is located off-card.*

A PROCESSOR WITH PERSONALITY

The DS2250 is a controller that integrates a lithium-backed RAM. This RAM can be partitioned as either program memory or data memory. Partitioning allows you to arrange the built-in memory in a variety of ways and even lets you use it in conjunction with external EPROMs and RAMs as you would with a standard 8031. Most implementations, however, do not use any external memory components since a full-blown system with program and data memory can be configured using just the DS2250. Not only does this make for a compact system in itself, but you retain all your bit-addressable I/O. This saves you parts and eliminates headaches since the on-chip I/O bits are flexible and easy to use.

Most controllers have no personality whatsoever until an executable program is placed in their code store. The situation is different with the DS2250, and you can carry on an intelligible dialog with the DS2250 before any program is transferred to the part whatsoever. The initial implanting of code can be performed with a standard parallel programming algorithm and using any 87C51 programming equipment. The DS2250 tolerates the programming voltages and understands the programming algorithms that these programmers use. Or it can be programmed directly from the host PC via the on-chip serial port with the aid of a built-in Loader ROM. This ROM is normally not accessible by the user, but by pulling RST high and \PSEN low, the DS2250 can be placed into bootstrap mode. When the chip is in this mode, the Loader ROM takes control of the DS2250 and performs a number of useful functions besides the initial loading of the executable image. Once bootstrap mode is exited and the DS2250 is released for operation, the Loader ROM relinquishes control and once again becomes transparent to the user having no effect on the memory map.

The Loader ROM understands and responds to a number of commands over the serial link. With the addition

Listing 1—*Support services for the PCF8583 I²C RTC/timer.*

```
#pragma LARGE CODE
#include "reg5000.h""

extern void RTC_IN(char length, char address, char *ptr);
extern void RTC_OUT(char length, char address, char *ptr);

/* Set the BCD RTC date and timeTime and date is at *ptr in the
form: milliseconds, seconds, minutes, hours, date, month */
void SetRTC(char *ptr)
{
    unsigned char i;
    unsigned char DataBuffer[7];
    DataBuffer[0] = 0x80;
    for (i = 1; i <= 6; i++)
        DataBuffer[i] = *ptr++;
    RTC_OUT(7, 0, DataBuffer);
    DataBuffer[0] = 0x0;
    RTC_OUT(1, 0, DataBuffer);
    return;
}

/* Get BCD RTC date and time. Time and date are returned at *ptr
as: milliseconds, seconds, minutes, hours, date, month */
void GetRTC(char *ptr)
{
    RTC_IN(6, 1, ptr);
    return;
}

/* Set BCD RTC date and time alarm. Alarm time and date are at
*ptr in the form: seconds, minutes, hours, date, month */
void SetAlarm(char *ptr)
{
    unsigned char i;
    unsigned char DataBuffer[8];
    DataBuffer[0] = 0x4;
    RTC_OUT(1, 0, DataBuffer);
    DataBuffer[0] = 0xf0;
    for (i = 1; i <= 5; i++)
        DataBuffer[i] = *ptr++;
    RTC_OUT(6, 8, DataBuffer);
    return;
}

/* Set the RTC interval timer where interval = 000 (no timer),
001 (milliseconds), 010 (seconds), 011 (minutes), 100 (hours), or
101 (days), and count = bcd up-count value */
void SetTimer(char interval, char count)
{
    unsigned char DataBuffer[2];
    DataBuffer[0] = 0x4;
    RTC_OUT(1, 0, DataBuffer);
    DataBuffer[0] = count;
    DataBuffer[1] =(interval | 0x8);
    RTC_OUT (2, 7, DataBuffer);
    return;
}

/* Assembler linkage: Input a string over the I2C bus */
static void RTC_IN(char length, char address, char *ptr)
{
    extern void Rec_I2C_String (void);
    #define RB0 ((char *) 0x10000L)
    RB0 [0] = length;
    B = address;
    ACC = 0xa0;
    DPTR = ptr;
```

(continued)

Listing 1—*continued*

```
    Rec_I2C_String();
    return;
}

/* Assembler linkage: Output a string over the I2C bus */
static void RTC_OUT(char length, char address, char *ptr)
{
    extern void Xmit_I2C_String(void);
    #define RB0 ((char *) 0x10000L)
    RB0 [0] = length;
    B = address;
    ACC = 0xa0;
    DPTR = ptr;
    Xmit_I2C_String();
    return;
}
```

of a PC hosted Initial Program Loader (IPL), a higher-level command shell is placed above the DS2250's command interpreter and the entire configuration, download, and verification operation can be set up to proceed automatically.

When the IPL is initially invoked to download a program, it builds a configuration file that is subsequently used to configure the DS2250 during the download sequence. Naturally, the most important information contained in this configuration file is the partition information that dictates where program memory ends and data memory begins.

Although ultimately very useful, the DS2250's partitioning capability is somewhat primitive compared to newer Dallas controllers such as the DS2251 and DS2252. The inherent limitation with the DS2250 is that, although separate program and data regions may be defined, they must be contiguous. That is, data memory must begin where program memory ends. Since the DS2250 only supports up to a maximum of 64K, this isn't really much of a problem. Certain applications could, however, benefit if it were possible to define overlapping program and data memories. In any case, once the program is loaded and the partition information is set, the program region is automatically protected. This protection results from the simple fact that the 8031 architecture provides no instructions that are capable of writing to program memory.

BRAIN TRANSPLANTS

Having the ability to download executables directly from a PC is a step in the right direction. However, if you're using the bootstrap loader for this purpose, you have to tolerate the Loader's particular data format and communications protocol (or lack thereof). In many cases, a more useful approach would allow program updating without the need for removing the unit from service. To accomplish downloading without unduly disrupting the installation, you would at least want to perform the operation within the confines of the system's native command syntax and communications protocol.

The designers of the DS2250 anticipated this need and provided the means of accomplishing a program download entirely under firmware control. Using the timed access method I described last month, you can gain access to the memory control register and define a new memory partition. This allows you to temporarily redefine program memory as data memory, which essentially grants you write access. Once you do this, data can be received from the system host in any format desired and can be written to data memory. On completion of the transfer, all that remains to be done is to partition the newly loaded region as program memory and begin executing the new program.

As a safety feature, when memory is partitioned under firmware control, the lowest 2K cannot be defined as data memory. This restriction is intended for your own safety. Needless to say, it would not bode well if you switched out all of program memory. Naturally, you must ensure that the program does not leave this 2K region until memory has been "normally" partitioned or you'll terminate your download real fast.

TWO BIT I/O

Last month I described how to connect to remotely located LCDs, keypads, and other general-purpose I/O devices using two wires over the I²C bus. The ec.25 uses these very same two wires for communicating with a number of local peripherals as well. Although the I²C card provides the bulk of the system I/O (both digital and analog), it also handles the system timekeeping and timing and provides two separate nonvolatile areas using RAM and E²PROM devices.

This card accommodates eight analog inputs and two analog outputs using two PCF8591s. A PCF8574 handles the digital I/O and provides eight bidirectional bits of digital I/O. Nonvolatile storage is contained in a 24C04 512-byte E²PROM. A PCF8583 contains a clock/calendar and a fairly elaborate timer subsystem as well as 256 bytes of battery-backed RAM. It is through the use of the timing facility contained in the PCF8583 that the ec.25 achieves its capacity for very long battery run times due to its capability for intermittent operation.

The schematic in Figure 1 shows the circuitry contained on the I²C card. U3, the PCF8583 RTC, being the only battery-backed peripheral on the card, gets its backup power from a source that is located off-card. Using an off-card backup source allows you to select the nonvolatile backup power at the system level. And having centralized backup power can lower costs, especially if you pick up a bunch of functions that ultimately need to have nonvolatile capabilities. This doesn't mean critical system peripherals such as large capacity RAM cards can't have local backup batteries, simply that it's optional.

I showed you last month how the power manager stepped down the

primary power using a regulator dedicated to providing backup power at 3.25 V. In an alternate configuration you may find yourself not needing to manage battery power at all and the system might be configured with a supply card that is optimized for line-powered operation. Naturally, if you don't have constant power in the first place, then the scheme I just outlined is of no use. The obvious solution would be to provide a centrally located backup power source such as a battery. Depending on the system's functional requirements and operational parameters, the backup source could be one of many different battery chemistries, either primary or secondary, or you could forego the battery and go with something such as a Supercap.

The system provides backup power at a nominal 3.25 V. You'd most likely want to operate somewhere around this level regardless of the backup method that you used. This backup power is combined with the main logic supply using mixing diodes (D2 and D3) fed into the V_{DD} pin of U3. U3's interrupt pin (\INT) can be driven by the RTC alarm or the timer alarm. As such, it can respond to a clock/calendar alarm and can function equally well as a fully programmable interval timer. Because of these capabilities, the PCF8583 proves to be well suited for providing the types of signaling necessary to control the ec.25's power sequencing.

As configured in the system, \INT connects directly to the power control circuitry located on the power manager card. The signals generated on \INT are controlled using firmware and can range from hundredths of a second to hundreds of days when running in timer mode. Due to this flexibility and the reasonably high repetition rates attainable, \INT is buffered by Q1 and Q2 and brought out to the DS2250 for general timing purposes. This signal can either be polled by the DS2250 or can function as an interrupt source.

Although the DS2250 is available with an integral RTC, I'm sure you can see why I elected to forego that particular feature based on what the PCF8583 offers. Even if I didn't need all the capabilities of the PCF8583, I'd have to think twice before giving in to the kind of clock that you get with the DS2250. Curiously, if you elect to use the DS2250T's built-in RTC, what you get is a DS1215 which is a "serial phantom" type of clock. This device resides in the same address space as other memory components, thus the name phantom. Enabling this particular clock requires a 64-bit pattern matching sequence that must be performed serially. Although I've used phantom devices in general and the DS1215 in particular with decent results, I've never been especially fond of them; there's just too much overhead. When you're stuck for an RTC in an existing design, they're great for getting you out of a jam. On the other hand, I find it perplexing that Dallas would actually incorporate something such as this into an original design. Maybe Dallas agrees with my opinion, since their newer microcontrollers (such as the DS2251) have abandoned this questionable approach and they now use the DS1283 instead which is a "standard" byte-wide device.

Listing 1 shows the fundamental support package for the PCF8583. Although only tapping a fraction of the PCF8583's capabilities, the functions contained within this program provide the basic services you'd require of an RTC and interval timer. Functions are shown that set the RTC, read the RTC, set the clock/calendar alarm, and set the timer alarm. Hopefully the

Listing 2—*Support services for the 24C04 I²C EEPROM.*

```
#pragma LARGE CODE
#include "reg5000.h"

extern void EEPROM_IN(char length, char address, char *ptr);
extern void EEPROM_OUT(char length, char address, char *ptr);

/* Write to the EEPROM, data is at *ptr. */
void WrEEPROM(char length, char address, char *ptr)
{
    EEPROM_OUT(length, address, ptr);
    return;
}
/* Read from the EEPROM, data is returned at *ptr. */
void RdEEPROM(char length, char address, char *ptr)
{
    EEPROM_IN(length, address, ptr);
    return;
}
/* Assembler linkage: Input a string over the I2C bus */
static void EEPROM_IN(char length, char address, char *ptr)
{
    extern void Rec_I2C_String(void);
    #define RB0 ((char *) 0x10000L)
    RB0 [0] = length;
    B = address;
    ACC = 0xa8;
    DPTR = ptr;
    Rec_I2C_String();
    return;
}
/* Assembler linkage: Output a string over the I2C bus */
static void EEPROM_OUT(char length, char address, char *XferPtr)
{
    extern void Xmit_I2C_String(void);
    #define RB0 ((char *) 0x10000L)
    RB0 [0] = length;
    B = address;
    ACC = 0xa8;
    DPTR = XferPtr;
    Xmit_I2C_String ();
    return;
}
```

and the second block at the selected address plus one. This might not sound like a big deal until you consider that some I²C E²PROMs get pretty big. In fact, the limiting factor seems to be the number of blocks that can be assigned based on the three programmable address pins allotted for this purpose. The 24C16, for example, defines 16K bits arranged as 8 blocks of 256 bytes. In this case, none of the programmable address bits are significant. If all your system needs is a bunch of E²PROMs, then you're fine, but beware that other peripherals may also be using the same base address as the big E²PROM. Listing 2 shows a minimal driver for the 24C04 and other I²C E²PROM devices.

The analog I/O is centered around two PCF8591 converters (U2 and U5). Each IC has four 8-bit analog inputs and one 8-bit analog output. All conversions are referenced to 2.5 V developed by R1 and VR1. Since the system is amenable to various configurations, battery power is monitored using channel 0 of the ADC rather than with a hardwired comparator.

The main power feed is derived from the 10-V preregulator when running from line power or from a battery when line power is not present. Using a PNP transistor (Q4) as a saturated switch, the main feed is attenuated through a divider composed of R13 and R14 and is presented to ADC channel 0. Q4 is switched on by Q3 only when V_{CC} is presented to the system in order to limit the battery's current drain and to safeguard the ADC when it is not powered.

Firmware support, shown in Listing 3, is provided for functions such as enabling and disabling the DACs, writing to the selected DAC, acquiring data from all ADCs, and acquiring data from a specified ADC. In order to conserve power, the DACs are defaulted off until they are turned on by program code. Since the DAC enable bit is transferred as part of the main control byte used for initiating any action in the analog subsection, a global variable containing the DAC enable mask bit is allocated for each channel. The enable and disable routines manipulate these mask bits.

Listing 3—*continued*

```
    return;
}
/* Get the ADC conversion for channels 0 through 7 */
void GetADCs(unsigned char *ptr)
{
    AD_IN(1, (0 | DacStatus[0]), ptr, 0);
    AD_IN(4, (0x5 | DacStatus[0]), ptr, 0);
    AD_IN(1, (0 | DacStatus[1]), ptr+4, 1);
    AD_IN(4, (0x5 | DacStatus[1]), ptr+4, 1);
    return;
}
/* Get the ADC conversion from the specified channel */
void GetADC(char *ptr, unsigned char channel)
{
    unsigned char chip;
    unsigned char address;
    unsigned char c = 2;
    if (channel < 4) {
        chip = 0;
        address = (channel | DacStatus[0]);
    }
    else {
        chip = 1;
        address = ((channel-4) | DacStatus[1]);
    }
    while (c--) AD_IN(1, address, ptr, chip);
    return;
}
/* Assembler linkage: Input a string over the I²C bus */
static void AD_IN(char length, char address, char *ptr,
                  unsigned char chip)
{
    extern void Rec_I2C_String(void);
    #define RB0 ((char *) 0x10000L)
    RB0 [0] = length;
    B = address;
    if (chip) ACC = 0x92;
       else ACC = 0x90;
    DPTR = ptr;
    Rec_I2C_String();
    return;
}
/* Assembler linkage: Output a string over the I²C bus */
static void AD_OUT(char length, char command, char *ptr,
                   unsigned char chip)
{
    extern void Xmit_I2C_String(void);
    #define RB0 ((char *) 0x10000L)
    RB0 [0] = length;
    B = command;
    if (chip) ACC = 0x92;
       else ACC = 0x90;
    DPTR = ptr;
    Xmit_I2C_String();
    return;
}
```

Setting a DAC's analog output consists of selecting the channel and passing a binary value. Acquiring data for all eight ADC channels consists of nothing more than setting up a destination pointer and calling the ADC service routine. Most of the work is performed by the I²C driver since it is capable of independently handling the string transfers over the I²C bus. Note that a dummy read is performed prior to the actual string transfer. This is because the PCF8591 uses the I²C clock as the conversion clock and, because of this, doesn't have the requested conversion data available

main functions are pretty self explanatory, but I should say a few words about the low-level assembler linkage.

I've already stated my unwillingness to monkey with the assembly level I²C driver. This month I'm calling the I²C string transfer routines, which means I have a couple of new registers to load up. DPTR along with the other SFRs are defined in a special header file so I don't have to worry about them. R0 and the other general-purpose registers, however, are not supported at the C level. Using a 24-bit pointer (RB0) explicitly defines register bank 0 as residing at location 0 and of residing in the data segment. When I want to set up R0, I end up moving the data byte indirectly to location 0 of RB0. Usually you work with your software products; sometimes you have to learn to work around them. If you're interested in looking at the PCF8583 from a couple of different angles, you might want to refer back to my columns in issues 35 and 42.

The 24C04 is a 4K-bit EEPROM. As is usual for parts of this type, data is organized in blocks of 256 bytes. The 24C04 has two such blocks. The 256-byte size is a result of the 8-bit addressing constraint inherent in the I²C implementation. If you have more than one block, essentially what you have to do to get around this restriction is handle each block as a sort of "virtual chip."

As you probably already know, all I²C peripherals have two components that make up their device address. One part of the address is fixed and is assigned on the basis of device type. There is also a programmable part that is selected by strapping address pins on the chip to either V_{CC} or V_{SS}. The ability to select the programmable part of the address is what allows you to have multiple chips that share the same base address on the same bus at the same time. What happens with the 24C04 is you give up one programmable address bit to accommodate the second 256-byte data block. Although A0 is tied to ground in the schematic (and you really must terminate it), it performs no address-related function. The 24C04 locates the first 256-byte block of data at its selected address

Listing 3—*Support services for the PCF8591 I²C ADC/DAC.*

```
#pragma LARGE CODE
#include "reg5000.h"

extern void AD_IN(char length,char address,char *ptr,char chip);
extern void AD_OUT(char length,char address,char *ptr,char chip);

unsigned char DacStatus[2];

/* Enable the selected DAC output */
void EnableDac(unsigned char channel)
{
    DacStatus[channel] = 0x40;
    return;
}
/* Disable the selected DAC output */
void DisableDac(unsigned char channel)
{
    DacStatus[channel] = 0;
    return;
}
/* Set selected DAC channel */
void SetDAC(unsigned char DacData, unsigned char channel)
{
    unsigned char chip;
    unsigned char DataBuffer[2];
    chip = channel;
    DataBuffer[0] = DacData;
    DataBuffer[1] = DacData;
    AD_OUT(2, DacStatus[channel], DataBuffer, chip);
```

(continued)

Listing 4—*Support functions for handling digital I/O over the I²C based byte-wide expansion port.*

```
#pragma LARGE CODE
#include "reg5000.h"

extern void DO_OUT (char c);
extern char DI_IN (void);

unsigned char DoMask;

/* Input the value of the DI port */
char GetDi (void)
{
    unsigned char c;

    c = DI_IN ();
    return (c);
}
/* Output the value to the DO port */
OutDo (char DoData)
{
    DoMask = DoData;
    DO_OUT (DoData);
    return;
}
/* Set the specified bits on the DO port */
SetDo (char DoData)
{
    DoMask |= DoData;
    DO_OUT (DoMask);
    return;
}
/* Clear the specified bits on the DO port */
ClearDo (unsigned char DoData)
{
    DoMask &= (~ DoData);
    DO_OUT (DoMask);
    return;
}
/* Assembler linkage: Output a byte over I2C bus */
static void DO_OUT (unsigned char c)
{
    extern void Xmit_I2C_Byte (void);

    ACC = 0x40;
    B = c;
    Xmit_I2C_Byte ();
    return;
}
/* Assembler linkage: Input a byte over I2C bus */
static char DI_IN (void)
{
    unsigned char c;
    extern void Rec_I2C_Byte (void);

    ACC = 0x40;
    Rec_I2C_Byte ();
    c = B;
    return (c);
}
```

until the current transfer is completed. That is, data moved on the initial transfer is always from the previous conversion and it is during this interval that the specified conversion takes place. Getting data from an ADC channel follows the same general procedure except that the chip and channel are isolated before the transfer is initiated. As before, two transfers are required to ensure that the selected channel is actually returned.

The PCF8574 handles the I²C-based bidirectional digital I/O. Listing 4 shows supported functions for reading from the port, writing to the port, set a bit (or bits), and clear a bit (or bits). You could just as well do these operations in-line rather that calling these support functions with their additional overhead. In some cases, though, it's a good idea to maintain control over even such seemingly trivial functions such as these. For instance, should I start accessing I/O from an interrupt level, it would behoove me to have a set of well-behaved routines that understand proper interrupt masking.

These drivers exemplify the need-to-do mode of operation. That is, they provide minimum levels of performance and functionality at a correspondingly minimal outlay of time and effort. They can always be tweaked if it turns out they are not up to the task. Surprisingly, it often turns out these tweaks are not required. This principle is closely related to the need-to-know principle which provides similar savings in time and energy.

Next month: BIOnet.

John Dybowski is an engineer involved in the design and manufacture of hardware and software for industrial data collection and communications equipment. He may be reached at john.dybowski@circellar.com.

SOURCES

For elements of this project, contact:

Mid-Tech Computing Devices
P.O. Box 218
Stafford Springs, CT 06075-0218
(203) 684-2442

Individual chips are available from

Pure Unobtainium
13109 Old Creedmoor Rd.
Raleigh, NC 27613
Phone/fax: (919) 676-4525

IRS

422 Very Useful
423 Moderately Useful
424 Not Useful